Binary Analysis Tool User and Developer Manual - describing version 23

Armijn Hemel – Tjaldur Software Governance Solutions

September 25, 2015

Contents

1 Introducing the Binary Analysis Tool **4**

2 Installing the Binary Analysis Tool **5**
2.1 Hardware requirements 5
2.2 Software requirements 5
2.2.1 Security warning 5
2.2.2 Installation on Fedora 5
2.2.3 Installation on Debian and Ubuntu 5
2.2.4 Installation on CentOS 6

3 Analysing binaries with the Binary Analysis Tool **6**
3.1 Running `bat-scan` 6
3.2 Interpreting the results 7
3.2.1 Output archive 7
3.2.2 XML output 8
3.2.3 Viewing results with `batgui` 8

4 Additional programs in the Binary Analysis Tool **9**
4.1 `busybox.py` and `busybox-compare-configs.py` 9
4.1.1 Extracting a configuration from BusyBox 9
4.1.2 Comparing two BusyBox configurations 9
4.2 `comparebinaries.py` 10
4.3 `sourcewalk.py` 10
4.4 `verifysourcearchive.py` 11
4.5 `findxor.py` 11

5 Binary Analysis Tool extratools collection **11**

A BAT scanning phases **12**
A.1 Identifier search 12
A.2 Pre-run checks 13
A.3 Unpackers 13
A.4 Leaf scans 14
A.5 Aggregators 15
A.6 Post-run methods 16

B Scan configuration **17**
B.1 Global configuration 17
B.1.1 `multiprocessing` and `processors` 17
B.1.2 `outputlite` 17
B.1.3 XML pretty printing 17
B.1.4 `tempdir` 18
B.1.5 `debug` and `debugphases` 18
B.1.6 `reporthash` 18
B.1.7 Global environment variables 18
B.2 Viewer configuration 18
B.3 Enabling and disabling scans 19
B.4 Blacklisting and whitelisting scans 19
B.5 Passing environment variables 19
B.6 Scan names 19
B.7 Scan conflicts 19
B.8 Storing results 20
B.9 Running setup code 20
B.10 Database configuration 20

C Analyser internals **21**
C.1 Code organisation 21
C.2 Pre-run methods 23
C.2.1 Writing a pre-run method 23
C.3 Unpackers 24
C.3.1 Writing an unpacker 24
C.3.2 Adding an identifier for a file system or compressed file . 24
C.3.3 Blacklisting and priorities 25
C.4 Leaf scans 25
C.4.1 Writing a leaf scan 25
C.4.2 Pretty printing for leaf scans 27
C.5 Aggregators 27
C.5.1 Writing an aggregator 27
C.6 Post-run methods 27
C.6.1 Writing a post-run method 28

D Building binary packages of the Binary Analysis Tool **28**
D.1 Building packages for RPM based systems from releases 28
D.2 Building packages for RPM based systems from Subversion . . . 28
D.2.1 Building `bat` 28
D.2.2 Building `bat-extratools` and `bat-extratools-java` . . 29
D.3 Building packages for DEB based systems from releases 29
D.4 Building packages for DEB based systems from Subversion 29
D.4.1 Building `bat` 29
D.4.2 Building `bat-extratools` and `bat-extratools-java` . . 30

E Binary Analysis Tool knowledgebase **30**
E.1 Generating the package list 30
E.2 Creating the database 31
E.3 License extraction and copyright information extraction 33
E.4 Converting the SQLite database to PostgreSQL 34

E.5 Database design . . . 34
E.5.1 `processed` table . . . 35
E.5.2 `processed_file` table . . . 36
E.5.3 `extracted_string` table . . . 36
E.5.4 `extracted_function` table . . . 36
E.5.5 `extracted_name` table . . . 37
E.5.6 `extracted_copyright` table . . . 37
E.5.7 `hashconversion` table . . . 37
E.5.8 `kernel_configuration` table . . . 38
E.5.9 `kernelmodule_alias` table . . . 38
E.5.10 `kernelmodule_author` table . . . 38
E.5.11 `kernelmodule_description` table . . . 38
E.5.12 `kernelmodule_firmware` table . . . 39
E.5.13 `kernelmodule_license` table . . . 39
E.5.14 `kernelmodule_parameter` table . . . 39
E.5.15 `kernelmodule_parameter_description` table . . . 39
E.5.16 `kernelmodule_version` table . . . 40
E.5.17 `licenses` table . . . 40
E.5.18 `renames` table . . . 40
E.5.19 `security_cert` table . . . 40
E.5.20 `security_cve` table . . . 41
E.5.21 `security_password` table . . . 41

F Identifier extraction and ranking scan **41**
F.1 Configuring identifier extraction . . . 41
F.2 Configuring the ranking method . . . 42
F.2.1 Interpreting the results . . . 44

G BusyBox script internals **44**
G.1 Detecting BusyBox . . . 45
G.2 BusyBox version strings . . . 45
G.3 BusyBox configuration format . . . 45
G.4 Extracting a configuration from a BusyBox binary . . . 46
G.4.1 BusyBox linked with uClibc . . . 46
G.4.2 BusyBox linked with glibc & uClibc exceptions . . . 47
G.5 Pretty printing a BusyBox configuration . . . 47
G.6 Using BusyBox configurations . . . 48
G.7 Extracting configurations from BusyBox sourcecode . . . 49

H Linux kernel identifier extraction **49**
H.1 Extracting visible strings from the Linux kernel binary . . . 50
H.2 Extracting visible strings from a Linux kernel module . . . 50
H.3 Extracting strings from the Linux kernel sources . . . 50
H.3.1 `EXPORT_SYMBOL` and `EXPORT_SYMBOL_GPL` . . . 50
H.3.2 `module_param` . . . 50
H.4 Forward porting and back porting . . . 50
H.5 Corner cases . . . 51

I Binary Analysis Tool performance tips **51**
I.1 Choose the right hardware 51
I.2 Use `outputlite` 52
I.3 Do not output results in XML 52
I.4 Use `AGGREGATE_CLEAN` when scanning Java JAR files 52
I.5 Disable `tmp` on tmpfs 52
I.6 Use tmpfs for writing temporary results 53

J Parameter description for default scans **53**
J.1 `compress` 53
J.2 `jffs2` 53
J.3 `lzma` 53
J.4 `tar` 54
J.5 `xor` 54
J.6 `file2package` 54
J.7 `findlibs` 54
J.8 `findsymbols` 54
J.9 `generateimages` 54
J.10 `identifier` 54
J.11 `licenseversion` 55
J.12 `prunefiles` 55
J.13 `hexdump` and `images` 55

K Default ordering of scans in BAT **55**
K.1 Pre-run scans 55
K.2 Unpack scans 56
K.3 Leaf scans 57
K.4 Aggregate scans 57

1 Introducing the Binary Analysis Tool

The Binary Analysis Tool (BAT) is a framework that can help developers and companies check binary files. Its primary application is for Open Source software license compliance, with a special focus on supply chain management in consumer electronics, but it can also be used for other checks of binary for example the presence of security bugs.

BAT consists of several programs written in Python. The most important program is the scanner for binary objects to unpack binaries recursively and apply a number of scans, for example for open source license compliance, visualising linking information, finding version information, and so on. There are also other programs to help with specific license compliance tasks, such as verifying if configurations for a given BusyBox binary match with the configuration in source code. Also included is a very experimental program to derive a possible configuration from a Linux kernel image, as well as programs to verify results from a binary scan with a source code archive.

2 Installing the Binary Analysis Tool

2.1 Hardware requirements

The tools in the Binary Analysis Tool can be quite resource intensive. They are largely I/O-bound (database access, reading files from disk), so it is better to invest in faster disks or ramdisks than in raw CPU power. Using more cores is also highly recommended, since most of the programs in the Binary Analysis Tool will vastly benefit from this.

2.2 Software requirements

To run BAT a recent Linux distribution is needed. Development is (currently) done on Fedora 21 and 22 and Ubuntu 14.04, so those platforms are likely to work best. Ubuntu versions older than 14.04, will not work due to a broken version of the PyDot package. Debian versions older than 7 are unsupported.

Versions older than Fedora 20 might not work if a database is used when scanning a whole directory of files instead of a single binary because of a bug in the version of matplotlib shipped on those distributions.

If the latest version from version control is used it is important to look at the file `setup.cfg` to get a list of the dependencies that should be met on the host system before installing BAT if the host system is Fedora. If the host system is Ubuntu or Debian this information will be in `debian/control`.

2.2.1 Security warning

Do not install BAT on a machine that is performing any critical functions for your organisation. There are certain pieces of code in BAT that have known security issues, such as some of the Squashfs unpacking programs in `bat-extratools` that have been lifted from vendor SDKs.

2.2.2 Installation on Fedora

To install on Fedora three packages are needed: `bat-extratools`, `bat-extratools-java` and `bat`. These can be downloaded from the BAT website in both prebuilt versions and as source RPM files. When installing the three files there should be a list of dependencies that should be installed to let BAT work successfully. Some of the dependencies are not in Fedora by default but need to be installed through external repositories, such as RPMfusion.

2.2.3 Installation on Debian and Ubuntu

To install on Debian and Ubuntu three packages are needed: `bat-extratools`, `bat-extratools-java` and `bat`. These can be downloaded from the BAT website as binary DEB files When installing the three files there should be a list of dependencies that should be installed to let BAT work successfully. Some of these packages are not in Debian by default but need to be installed by enabling extra repositories such as Debian `non-free`.

2.2.4 Installation on CentOS

In some cases it is possible to run BAT on CentOS (6.6 or 7 has been tested with) but some functionality will not be available, such as UBI/UBIFS unpacking and the scans creating graphs with PyDot (ELF linking, kernel module linking).

It might be necessary to enable the EPEL repository (`https://fedoraproject.org/wiki/EPEL`) as well as RepoForge. A few packages might have to be installed manually. To rebuild `bat-extratools-java` a newer version of Java might be required.

3 Analysing binaries with the Binary Analysis Tool

BAT consists of several programs and a few helper scripts (not meant to be used directly). The main purpose of the Binary Analysis tool is to analyse arbitrary binaries and review results. Analysis of the binary is done via a commandline tool (`bat-scan`), while the results can be viewed using a special graphical viewer (`batgui`).

3.1 Running `bat-scan`

The `bat-scan` program can scan in two modes: either scan a single binary, or scan a whole directory of files.

To scan a single binary you will need to supply three parameters to `bat-scan`:

1. `-c` : path to a configuration file
2. `-b` : path to the binary to be scanned
3. `-o` : path to an output file, where unpacked files, reports, plus the final program state be written to. This file can later be opened with the viewer.

The default install of BAT comes with a configuration file (installed in `/etc/bat/` although this will likely change in the future) with default settings that have proven to work well but almost everything can be changed or tweaked. A lengthy explanation of the different types of scans and their configuration can be found in the appendix.

A typical invocation looks like this:

```
python bat-scan -c /path/to/configuration -b /path/to/binary -o
/path/to/outputfile
```

XML output, if enabled in the configuration file, will be written to standard output. Any debugging messages or error messages will appear on standard error.

To scan a directory you will need to supply three parameters to `bat-scan`:

1. `-c` : path to a configuration file
2. `-d` : path to a directory with files to be scanned
3. `-u` : path to a directory where output files will be written to

For example:

```
python bat-scan -c /path/to/configuration -d /path/to/dirwithbinaries
-u /path/to/dirwithoutputfiles
```

The format of output files in "directory scan" mode will be the name of the original file with the suffix `.tar.gz`.

3.2 Interpreting the results

There are two formats in which `bat-scan` can output its results:

1. archive file containing program state, complete unpacked directory tree containing all unpacked data (unless `outputlite` was set to `yes`), plus possibly some extra generated data, such as pictures and more reporting. These dumps are meant to be used by `batgui`. This is the default format.
2. XML file (optional/configurable). This format is deprecated and will likely be removed in the near future.

3.2.1 Output archive

The output archive contains a few files and directories (depending on scan configuration):

- `scandata.pickle` - Python pickle containing information about the structure of the binary, including offsets, paths, tags, and so on. It does not contain any of the actual scan results.
- `scandata.json` - JSON file containing a subset of the information in `scandata.pickle`. This file is only generated if the `generatejson` scan is enabled.
- `data` - directory containing the full unpacked directory tree. If `outputlite` is set to `yes` this directory will be omitted.
- `filereports` - directory containing Python pickle files (gzip compressed) with scan results. Since identical files might be present the results are stored per checksum, not file name.
- `images` - directory containing various images with results of scans (depending on which scans are enabled), per checksum
- `offsets` - directory with gzip compressed Python pickle files containing the offsets of possible file systems, compressed files and media files found in the file. This directory as well as its files will only be created if `dumpoffsets` is set to `yes` in the global configuration.
- `reports` - directory containing HTML and (optionally) JSON reports, per checksum

3.2.2 XML output

If configured `bat-scan` outputs its results in XML format on standard output. After redirecting the output to a file it is possible to look at this file with a commandline tool such as `xml_pp` or a webbrowser such as Mozilla Firefox.

This XML file is not meant for human consumption, but for use by for example reporting tools. A word of warning is needed: the XML format is not very well designed and not well maintained and it will likely be removed in the future. If you want to programmatically process information from BAT it is recommended to use the Python pickles (or optionally JSON output) to extract information instead. In the BAT source code repository a file documenting these pickles can be found.

The default XML pretty printer as shipped by BAT outputs an XML file that starts with metadata, such as:

- date, plus time of the scan (local time of the computer, in UTC)
- name of the file
- SHA256 cryptographic checksum of the file, uniquely identifying it
- size of the file
- filetype as determined by `file` on a Linux system
- relative path inside the unpacked system, plus the absolute path inside the file system, which is useful for later analysis

If any of the scans were successful the results of the these scans can be found in the element `scans`.

For each successful unpack action the following attributes are reported:

- name of the scan (corresponding to the name of the scan in the configuration file)
- offset in the parent file of the compressed file, file system or media file

3.2.3 Viewing results with `batgui`

The `batgui` program was made to view the results of the analysis process easily, without having to dig through XML. The viewer has two modes: simple and advanced. In simple mode a tree of the unpack results will be shown, and each file in the tree can be clicked to display more information. Depending on which scans were run the tree will be decorated with more information, such as the type of the file (based on tags), or if matches were found with the ranking method. Using a filtering system (available from the menu) files that are typically uninteresting for license compliance engineering (empty files/directories, symbolic links, graphics files and so on) can be ignored.

Information that is shown per file depends on the scans that were run and the type of file. For most files information like size, type, path (both relative inside the unpacked binary, as well as absolute in the scanning tree) will be shown. If the ranking method was enabled results of the ranking process such as matched strings, function names, a license guess etecetera will be displayed as well.

In the optional advanced mode more results will be shown, such as a graphical representation of a file, where every bit in the binary has been assigned a grayscale value, plus a textual representation of a file generated with `hexdump`. Advanced mode is disabled by default, since loading the additional pictures and data is quite resource intensive and it will only be useful in very specific cases. It also requires that these special files are generated by BAT when scanning a file. This is not done by default but needs an explicit configuration change. Advanced mode might be removed from the GUI in future versions of BAT.

4 Additional programs in the Binary Analysis Tool

4.1 `busybox.py` and `busybox-compare-configs.py`

Two other tools in BAT are `busybox-compare-configs.py` and `busybox.py` (in the subdirectory `bat`). These two tools are specifically used to analyse BusyBox binaries. BusyBox is in widespread use on embedded devices and the license violations of BusyBox are actively enforced in court.

BusyBox binaries on embedded machines often have different configurations, depending on the needs of the manufacturer. Since providing the correct configuration is one of the requirements for license compliance it is important to be able to determine the configuration of a BusyBox binary and verify that there is a corresponding configuration file in the source code release.

The BusyBox processing tools in BAT try to extract the most likely configuration from the binary and print it in the right format for that version of BusyBox.

`busybox.py` is used to extract the configuration from a binary. Afterwards `busybox-compare-configs.py` can be used to compare the extracted configuration with a vendor supplied configuration.

4.1.1 Extracting a configuration from BusyBox

Extracting a configuration from a BusyBox executable is done using `busybox.py` which can be found in the `bat` directory. It needs two commandline parameters: the path to the binary and the path to a directory containing a directory `configs` which has files containing mappings from BusyBox applet names to BusyBox configuration directives. By default this value is hardcoded as `/etc/bat`, but this might change in the future.

Output (a configuration) is written to standard output.

```
python bat/busybox.py -b /path/to/busybox/binary -c
     /path/to/pre/extracted/configs > /path/to/saved/config
```

This command will save the configuration to a file, which can be used as an input to `busybox-compare-configs.py`.

4.1.2 Comparing two BusyBox configurations

After extracting the configuration the extracted configuration can be compared to another configuration, for example a configuration as supplied by a vendor in a source code archive:

```
python busybox-compare-configs.py -e /path/to/saved/config
    -f /path/to/vendor/configuration -n $version
```

4.2 comparebinaries.py

The `comparebinaries.py` program compares two file trees with for example unpacked firmwares. It is intended to find out which differences there are between two binaries (like firmwares) unpacked with BAT.

There are two scenarios where this program can be used:

1. comparing an old firmware (that is already known and which has been verified) to a new firmware (update) and see if there are any differences.
2. comparing a firmware to a rebuild of a firmware as part of compliance engineering.

A few assumptions are made:

1. both firmwares were unpacked using the Binary Analysis Tool
2. files that are in the original firmware, but not in the new firmware, are not reported (example: removed binaries). This will change in a future version.
3. files that are in the new firmware but not not in the original firmware are reported, since this would mean additions to the firmware which need to be checked.
4. files that appear in both firmwares but which are not identical are checked using `bsdiff` to determine the size of the difference.

With checksums it is easy to find the files that are different. Using `bsdiff` it becomes easier to prioritise based on the size of the difference.

Small differences are probably not very interesting at all:

1. time stamps (BusyBox, Linux kernel, and others record a time stamp in the binary)
2. slightly different build system settings (home directories, paths, and so on).

Bigger differences are of course much more interesting.

4.3 sourcewalk.py

This program can quickly determine whether or not source code files in a directory can be found in known upstream sources. It uses a pregenerated database containing names and checksums of files (for example the Linux kernel) and reports whether or not the source code files can be found in the database based on these checksums.

The purpose of this script is to find source code files that cannot be found in upstream sources to reduce the search space during a source code audit.

This script will not catch:

- binary files
- patch/diff files
- anything that does not have an extension from the list in the script
- configuration files/build scripts

4.4 `verifysourcearchive.py`

The `verifysourcearchive.py` program is to verify a source code archive using the result of a scan done with BAT.

4.5 `findxor.py`

The `findxor.py` program can be used to find possible XOR "encryption" keys. It prints the top 10 (hardcoded limit) of most common byte sequences (16 bytes) in the file. These can then be added to the `batxor.py` module in BAT. This will likely change in the future.

5 Binary Analysis Tool extratools collection

To help with unpacking non-standard file systems, or standard file systems for which there are no tools readily available on Fedora or Ubuntu there is also a collection of tools that can be used by BAT to unpack more file systems. These tools are not part of the standard distribution, but have to be installed separately. They are governed by different license conditions than the core BAT distribution.

Currently the collection consists of:

- `bat-minix` has a Python script to unpack Minix v1 file systems that are frequently found on older embedded Linux systems, such as IP cameras.
- modified version of `code2html` (which is unmaintained by the upstream author) that adds support for various more programming languages.
- unmodified version of `simg2img` needed for converting Android sparse files to ext4 file system images.
- unmodified version of `romfsck` needed for unpacking romfs file systems.
- modified version of `cramfsck` that enables unpacking cramfs file systems.
- unmodified version of `unyaffs` that enables unpacking for some (but not all) YAFFS2 file systems.
- various versions of `unsquashfs` that enable unpacking variants of SquashFS. These versions have either been lifted from vendor SDKs, the OpenWrt project, or upstream SquashFS project.
- `ubi_reader` is a set of tools to deal with UBI/UBIFS images.
- `bat-visualisation` containing a few custom tools to help generate pictures. These might be removed in the future.

- two Java projects: `jdeserialize` and `ddex`, to help respectively with unpacking serialized Java files and scanning binary files from the Dalvik VM (Android).

The collection is split in three packages: `bat-extratools-java` contains the two Java packages, the `ubi_reader` package contains UBI/UBIFS specific tools, the `bat-extratools` package contains the rest.

A BAT scanning phases

BAT uses a brute force approach for analysing a binary. It assumes no prior knowledge of how a binary is constructed or what is inside the binary. Instead it tries to determine what is inside by applying a wide range of methods, such as looking for known identifiers of file systems and compressed files and running external tools to find contents in the binary. It should be noted that there are possibilities to add more information to the system to speed up scanning.

During scanning of a file the following steps are taken:

1. identifier search, using a list of known identifiers
2. verifying file type of a file and, if successful, tagging it. Tags can be used later on to give more information to the scanner.
3. unpacking file systems, compressed files and media files from the file, carving them out of the file first.
4. repeat steps 1 - 3 for each file that was unpacked in step 3
5. run individual scans on each file if no further unpacking is possible
6. optionally aggregate scan results or modify results based on information that has become available during the scan
7. process results from scans in step 5 and 6 and generate reports
8. pack results into an archive that can be used by the viewer application or other applications

A.1 Identifier search

The first action performed is scanning a file for known identifiers of compressed files, file systems and media files. The identifers are important for a few reasons: first, they are used to determine which checks will run. They are also used frequently throughout the code for verification and speeding up unpacking. If a scan depends on a specific identifier being present it can be set using the `magic` attribute in the configuration. If an identifier is not defined anywhere in the configuration file as needed it will be skipped during the identifier search to speed up the identifier search. Some scans define an additional magic header in `optmagic`. The values defined in `optmagic` are not authoritive, but should be treated as hints. A good example is the YAFFS2 scan.

The marker search cannot be enabled or disabled via the configuration file. The markers that are searched for are found in `bat/fsmagic.py`.

As an optimization the marker search can be skipped for some files if they have an extension which gives a possible hint about what kind of file it might be. For example, for gzip compressed files files with the extension `gz` a special method (configured in the configuration for the gzip unpacker) is first run to see if the file is actually a gzip file, without looking at any other markers, or trying other scans first.

As a further optimization there is one method that is run for ZIP-files (including Android APK files and Java JAR, EAR and WAR files) before the generic marker search: large firmwares tend to be distributed as ZIP-files. By quickly checking if a file is a complete ZIP-file time can be saved. This method will be removed in the near future and rewritten to a similar method as is used for gzip.

If multiple CPUs are available and the top level file is larger than a certain limit and does not have a known extension as described above the marker search will be done in parallel as a speed up. The limit can be set in the global configuration using the variable `markersearchminimum`. The default value for this variable is 20 million bytes.

A.2 Pre-run checks

Before files are unpacked they are briefly inspected and if possible tagged. Tags are used to pass hints to methods that are run later to avoid unnecessarily scanning a file and to reduce the amount of false positives.

For example, files that only contain text are tagged as `text`, all other files are tagged as `binary` (this depends on the implementation of Python. Python 2 only considers (by default) ASCII to be valid text). Methods that only work on binaries can then ignore anything that has been tagged as `text`.

Other checks that are available are for valid XML, various Android formats, ELF executables and libraries, graphics files, audio files, and so on.

The prerun checks can easily be identified in the configuration, since it has its type set to `prerun`:

```
[verifytext]
type        = prerun
module      = bat.prerun
method      = verifyText
priority    = 3
description = Check if file contains just ASCII text
enabled     = yes
```

Prerun verifiers can optionally make use of tags that are already present by using `magic` and `noscan` attributes, which will be explained in detail later for the unpackers.

A.3 Unpackers

Unpackers can be recognized in the configuration because their type is set to `unpack`, for example:

```
[jffs2]
type        = unpack
```

```
module      = bat.fwunpack
method      = searchUnpackJffs2
priority    = 2
magic       = jffs2_le:jffs2_be
noscan      = text:xml:graphics:pdf:compressed:audio:video:mp4:elf:java:resource:dalvik
description = Unpack JFFS2 file systems
enabled     = yes
```

In BAT 23 the following file systems, compressed files and media files can be unpacked or extracted:

- file systems: Android sparse files, cramfs, ext2/ext3/ext4, ISO9660, JFFS2, Minix (specific variant of v1 often found on older embedded Linux systems), SquashFS (several variants), romfs, YAFFS2 (specific variants), ubifs (not on all systems)
- compressed files and executable formats: 7z, ar, ARJ, BASE64, BZIP2, compressed Flash, CAB, compress, CPIO, EXE (specific compression methods only), GZIP, InstallShield (old versions), LRZIP, LZIP, LZMA, LZO, MSI, pack200, RAR, RPM, RZIP, serialized Java, TAR, UPX, XZ, ZIP (including APK, EAR, JAR and WAR)
- media files: GIF, ICO, PDF, PNG, WOFF, CHM

Most of the unpackers for these file systems, compressed files and media files are located in the file `bat/fwunpack.py`.

Unpacking differs per file type. Most files use one or more identifiers that can be searched for in a binary blob. Using this information it is possible to carve out the right parts of a binary blob and verify if it indeed contains a compressed file, media file or file system.

There is not always an identifier that can be searched for. The YAFFS2 file system layout for example is dependent on the hardware specifics of the underlying flash chip. Without knowing these specifics it is not possible to specifically search for a valid YAFFS2 file system. This scan therefore tries to run on every file, unless explicitely filtered out (using `noscan` and tags).

Other file types (such as ARJ files) have a very generic identifier, so there are a lot of false positives. This causes a big increase in runtime. The ARJ unpacker is therefore disabled by default.

LZMA is another special case: there are many different valid headers for LZMA files, but in practice only a handful are used.

If unpacking is successful a directory with unpacked files is returned, and, if available, some meta information to avoid duplicate scanning (blacklisting information and tags). The unpacked files are added to the scan queue and scanned recursively.

A.4 Leaf scans

Leaf scans are scans that are run on every single file after unpacking, including files that contained files that were found and extracted by unpackers.

Leaf scans can be recognized in the configuration because their type is set to `leaf`, for example:

```
[markers]
type        = leaf
module      = bat.checks
method      = searchMarker
noscan      = text:xml:graphics:pdf:compressed:audio:video
description = Determine presence of markers of several open source programs
enabled     = yes
```

The current leaf scans that are available in BAT are:

- marker scan searching for signature scans of a few open source programs (dproxy, ez-ipupdate, hostapd, iptables, iproute, libusb, loadlin, RedBoot, U-Boot, vsftpd, wireless-tools, wpa-supplicant)
- advanced search mode using ranking of strings, function names, variable names, field names and Java class names using a database (for ELF and Java, both regular JVM and Dalvik)
- BusyBox version number
- dynamic library dependencies (ELF files only)
- file architecture (ELF files only)
- Linux kernel module license (Linux kernel modules only)
- Linux kernel version number, plus detection for several subsystems
- PDF meta data extraction
- presence of URLs indicating an open source license
- presence of URLs indicating forges/collaborative software development sites (SourceForge, GitHub, etcetera)

The fast string searches are meant for quick sweep scanning only. They have their limits, can report false positives or fail to identify a binary. They should only be used to signal that further inspection is necessary. For a thorough investigation the advanced search mode should be used. These scans are likely to be disabled in the future in the default configuration.

A.5 Aggregators

Sometimes it helps to aggregate results of a number of files, or it could be useful to perform other actions after all the individual scans have run. The best example is dealing with JAR-files (Java ARchives). Individual Java class files often contain too little information to map them reliably to a source code package.

Typically a class file contains just a few method names, or field names, or strings. If inner classes are used it can be even worse and information from a single source code file could be scattered across several class files.

Since Java programs (note: excluding Android) are typically distributed as a JAR that is either included at runtime or directly executed, similar to an ELF library or ELF executable, it makes perfect sense to treat the JAR file as

a single unit and aggregate results for the individual class files and assign them to the JAR file.

Aggregators take all results of the entire scan as input.

Currently the following aggregators are available:

- advanced identifier search and classification
- aggregating result of individual Java class files in case they come from the same JAR file.
- cleaning up/fixing results of duplicate files: often firmwares contain duplicate files. Sometimes some more information is available to make a better choice as to which file is the duplicate and which one is the original version
- checking dynamically linked ELF files
- finding duplicate files
- finding licenses and versions of strings and function names that were found and optionally pruning the result set to remove unlikely results.
- pruning files from the scan completely if they are not interesting (such as pictures, or text files) using tags.
- generating pictures of results of a scan
- generating reports of results of a scan

A.6 Post-run methods

In BAT there are methods that are run after all the regular work has been performed, or "post-run". These methods should not alter the scan results in any way, but just use the information from the scanning process. A typical use case would be to present the data in a nicer to use format than the standard report, to use more external data sources or generate graphical representations of data.

The post-run methods have the type `postrun` in the configuration, for example:

```
[hexdump]
type        = postrun
module      = bat.generatehexdump
method      = generateHexdump
noscan      = text:xml:graphics:pdf:audio:video
envvars     = BAT_REPORTDIR=/tmp/images:BAT_IMAGE_MAXFILESIZE=100000000
description = Create hexdump output of files
enabled     = no
storetarget = reports
storedir    = /tmp/images
storetype   = -hexdump.gz
cleanup     = no
```

B Scan configuration

The analysis process is highly configurable: methods can be simply enabled and disabled, based on need: some methods can run for quite a long time, which might be undesirable at times. Configuration is done via a simple configuration file in Windows INI format.

Most sections are specific to scanning methods, except two sections: a global section and one section specific for the viewer tool.

B.1 Global configuration

The global configuration section is called `batconfig`. In this section various global settings are defined. The section looks like this:

```
[batconfig]
```

B.1.1 `multiprocessing` and `processors`

The `multiprocessing` configuration option determines whether or not multiple CPUs (or cores) should be used during scanning. The default configuration as shipped in the official BAT distribution is to use multiple threads:

```
[batconfig]
multiprocessing = yes
```

If set to `yes` the program will start an extra process per CPU that is available for parts of the program that can be run in parallel. In most cases it is completely safe to use multiprocessing.

It might be desirable to not use all processors on a machine, for example if there are multiple scans of BAT running at the same time, or if other tasks need to run on the machine. It is possible to set the maximum amount of processors to use with the `processors` option:

```
processors = 2
```

B.1.2 `outputlite`

Another setting in this section is `outputlite`:

```
outputlite = yes
```

It defaults to `yes`. If set to `yes` the output archive will omit a full copy of the unpacked data, significantly decreasing the size of the output archive, but making it harder to do a "post mortem" on the unpacked data (a new analysis should be run to get it again).

B.1.3 XML pretty printing

There are two settings that determine where the code of the optional XML pretty printer can be found:

```
module = bat.simpleprettyprint
output = prettyprintresxml
```

These two settings should always be used together.

B.1.4 `tempdir`

There is one setting to set the prefix for creating temporary files or directories, namely `tempdir`. By default the directory for creating temporary files and directories is `/tmp`. There might be situations where the temporary directory might need to be changed, for example for unpacking on a faster medium (ramdisk, SSD) than a normal harddisk. It can be used as follows:

```
tempdir = /ssd/tmp
```

B.1.5 `debug` and `debugphases`

To assist in debugging and finding errors in scans of BAT there are two settings: `debug` and `debugphases`. The setting `debug` can be used to enable and disable debugging. If set multiprocessing will be disabled and information about which file is scanned and which method is run will be printed on standard error. If specified without `debugphases` this will apply to all scan phases. The `debugphases` parameter can be used to limit this behaviour to just one or a few phases. The other phases will behave normally. For example, this will enable debugging, but just for the leaf scans and aggregate scans:

```
debug = yes
debugphases = leaf:aggregate
```

B.1.6 `reporthash`

If `reporthash` is set, then hashes in the ranking scan that come from the BAT database will be converted from SHA256 (default) to the hash if supported (currently MD5, SHA1 and CRC32 are supported) in the default BAT database as shipped by Tjaldur Software Governance Solutions:

```
reporthash = sha256
```

B.1.7 Global environment variables

Since BAT 20 it is possible to supply global environment variables. These can be shared between scans. They can be overridden by individual scans. For example to set the environment variable `BAT_NAMECACHE_C` for all scans you would put something like this in the global configuration:

```
envvars      = BAT_NAMECACHE_C=/home/bat/db/functioncache_c
```

As a rule of thumb: settings that are shared between all scans (such as the location of various databases) should be set in the global sections, while scan specific options should be in the scan specific sections.

B.2 Viewer configuration

The other global section is `viewer`. This section is specific for the graphical frontend and is not used in any other parts of BAT and might be moved to a separate configuration file in a future version of BAT.

B.3 Enabling and disabling scans

The standard configuration file enables most of the scans and methods implemented in BAT by default. Scans can be enabled and disabled by setting the option `enabled` to `yes` and `no` respectively.

Another way to not run a scan is to outcomment the entry in the configuration file (by starting the line with the `#` character), or by removing the section from the configuration file.

B.4 Blacklisting and whitelisting scans

Files can be explicitely blacklisted for scanning by using the `noscan` configuration setting. The value of this parameter is a list of tags, separated by colons:

```
noscan      = text:xml:graphics:pdf:audio:video
```

Similarly files can be whitelisted by using the `scanonly` setting. Only files that are tagged with any of the values in this list (if not empty) will be scanned. If there is an overlapping value in `scanonly` and `noscan` then the file will not be scanned.

B.5 Passing environment variables

All scans have an optional parameter `scanenv` defaulting to an empty Python dictionary. In the configuration file a colon separated list of name/value pairs can be specified using the keyword `envvars`. These will then become available in the environment of the scan:

```
envvars     = BAT_REPORTDIR=/tmp/images:BAT_IMAGE_MAXFILESIZE=100000000
```

If the environment of a scan needs to be adapted in the context of a single file it is important to first make a copy of the environment or the environment might be modified for the scan for all other files that are scanned.

B.6 Scan names

The name of the scan is used in various places, for example for storing results or for determining scan conflicts. The `name` parameter can be used to set the name for the scan. If no name is specified the name of the section of the scan is used instead.

```
name = gzip
```

B.7 Scan conflicts

Possibly scans can conflict with other scans in the same phase and they should not be enabled at the same time. To indicate that a scan conflicts with others the `conflicts` option can be set:

```
conflicts = gzip:bzip2
```

If there is a conflict in the configuration BAT will refuse to run. Currently BAT only looks at conflicts in the same unpacking phase and only for scans that are enabled.

B.8 Storing results

Postrun scans and aggregate scans that output data, for example graphics files or reports, can specify which files should be added to the output file. There are three settings that should be set together:

```
storetarget = images
storedir    = /tmp/images
storetype   = -piechart.png:-version.png
```

The `storetarget` setting specifies the relative directory inside the output TAR archive. The `storedir` setting tells where to get the files that need to be stored can be found (this should be where the postrun scan or aggregate scan stores its results). The `storetype` setting is a colon separated list of extensions/partial file names that the files should end in (typically the rest of the filename is a SHA256 value).

The additional setting `cleanup` can be used to instruct BAT that the files generated by this postrun scan or aggregate scan should be removed after copying them into the result archive:

```
cleanup     = yes
```

The `cleanup` setting should be set to `yes` unless the results do not change in between subsequent runs of BAT.

Currently (BAT 22) if `cleanup` is set the files are written directly to output directories. The values of these directories are hardcoded (and match values that the GUI expects) but these will be replaced by the value of `storetarget` in a later release.

B.9 Running setup code

For some scans it is necessary to run some setup code to ensure that certain conditions are met, for example that databases exist, or that locations are readable/writeable. These checks only need to be run once. Based on the result of the setup code the scan might be disabled if certain conditions are not met.

There is a special hook for leaf scans to run setup code for the scan:

```
setup     = nameOfSetupMethod
```

The files `bat/identifier.py` and `bat/licenseversion.py` contain very extensive examples of setup hooks.

B.10 Database configuration

Currently two database engines are supported by BAT: SQLite and PostgreSQL. Various scans can use a database backend. Depending on the scan or set up the databases can be in SQLite format, or PostgreSQL format and these can be freely mixed but it is advised to use one database backend.

The database backend can be chosen either per scan, or defined as a global environment variable.

To select SQLite as a backend set `dbbackend` as follows:

```
dbbackend = sqlite
```

or for PostgreSQL:

```
dbbackend = postgresql
```

If PostgreSQL is chosen a few other variables have to be set (username, password, database):

```
postgresql_user     = bat
postgresql_password = bat
postgresql_db       = bat
```

Optionally a port and host can be set too:

```
postgresql_host     = 127.0.0.1
postgresql_port     = 5432
```

Depending on the version of `python-psycopg2` it could be that `postgresql_host` and `postgresql_port` both have to be specified. On CentOS 6.x both have to be set.

C Analyser internals

The analyser was written with extensibility in mind: new file systems or variants of old ones tend to appear regularly (for example: there are at least 5 or more versions of SquashFS with LZMA compression out there).

C.1 Code organisation

`bat-scan` is merely a frontend for the real scanner and only handle the list of scans, the binary/binaries to scan and where to write the output file(s).

The meaty bits of the analyser can be found in files in the `bat` subdirectory (note that this directory currently contains more files than are actually used by BAT at the moment):

- `batdb.py` contains the BAT database abstraction code, as well as a query rewriting method.
- `batxor.py` contains experimental code to deal with files that have been obfuscated with XOR.
- `bruteforcescan.py` contains the main logic of the program: it launches scans based on what is inside the binary and the scans that are enabled, collects results from scans and writes results to an output file.
- `busybox.py` and `busyboxversion.py` contain code to extract useful information from a BusyBox binary, such as the version number.
- `checks.py` contains various leaf scans, like scanning for certain marker strings, or the presence of license texts and URLs of forges/collaborative software development sites.

- `ext2.py` implements some functionality needed for unpacking ext2 file systems.
- `extractor.py` provides convenience functions that are used throughout the code.
- `file2package.py` has code to match names of files to names of packages from popular distributions in a database.
- `findduplicates.py` is used to find duplicate files in the scanned archive.
- `findlibs.py` and `interfaces.py` are for researching dynamically linked ELF files in the archive.
- `findsymbols.py` is for researching relationships between Linux kernel modules and the Linux kernel in the archive, specifically for the declared licenses and the license of symbols used.
- `fixduplicates.py` is used to correct tagging of files that were tagged incorrectly as duplicates, as they are the original, not the copy. For now this is only for ELF files.
- `fsmagic.py` contains identifiers of various file systems and compressed files, like magic headers and offsets for which might need to be corrected.
- `fwunpack.py` includes most of the functionality for unpacking compressed files and file systems.
- `generatehexdump.py` and `images.py` generate textual and graphical representations of the input files.
- `generatereports.py`, `generateimages.py`, `guireport.py`, `generatejson.py` and `piecharts.py` generate textual and graphical representations of results of the analysis.
- `identifier.py` implements functionality to extract identifiers (string constants, function names, method names, variable names, and so on) from binary files and make them available for further analysis.
- `jffs2.py` has code specific to handling JFFS2 file systems.
- `kernelanalysis.py` includes code to extract information from Linux kernel images and Linux kernel modules.
- `kernelsymbols.py` is used for generating dependency graphs for Linux kernel modules and indicating any possible license issues of exported symbols and declared licenses.
- `licenseversion.py` gets version and licensing information for uniquely identified strings and function names (and in the future variable names too) from the database. It can optionally prune the result set to only include relevant versions. It also contains code to aggregate results of Java class files from a JAR file and assign results to the JAR file instead of the individual class files.

- `prerun.py` contains scans that are run in the pre-run phase for correctly tagging files as early in the process as possible.
- `prunefiles.py` can be used to remove files with a certain tag from the scan results. This is useful for for example graphics files.
- `renamefiles.py` is used for renaming files to use a more logical name after more contextual information from the scan has become available. For example: detect an initramfs in the Linux kernel and rename the temporary file to `initramfs`.
- `security.py` contains several security scans.
- `simpleprettyprint.py` has the default XML prettyprinter.
- `unpackrpm.py` has code specifically for unpacking RPM archives.

C.2 Pre-run methods

Pre-run methods check and tag files, so the files can be ignored by later methods and scans, reducing scanning time and preventing false positives. While tagging is not exclusive to pre-run methods it is their main purpose.

C.2.1 Writing a pre-run method

Pre-run methods have a strict interface. Parameters are:

- `filename` is the absolute path of the file that needs to be tagged
- `tempdir` is the (possibly) empty name of a directory where the file is. This is currently unused and might be removed in the future.
- `tags` is the set of tags that have already been defined for the file.
- `offsets` is the set of offsets that have been found for the file
- `scanenv` is an optionally empty dictionary of environment variables that can be used to pass extra information to the pre-run method.
- `debug` is an environment variable that can be used to optionally set the scan in debugging mode so it can print more information on standard error. By default it is set to `False`.
- `unpacktempdir` is the location of a directory for writing temporary files. This value is optional and by default it is set to `None`.

Return values are:

- a list containing tags

Example:

```
def prerunMethod(filename, tempdir=None, tags=[], offsets={},
                 scanenv={}, debug=False, unpacktempdir=None):
        newtags = []
        newtags.append('helloworld')
        return newtags
```

C.3 Unpackers

Unpackers are responsible for recursively unpacking binaries until they can't be unpacked any further.

C.3.1 Writing an unpacker

The unpackers have a strict interface:

```
def unpackScan(filename, tempdir=None, blacklist=[], offsets={},
               scanenv={}, debug=False):
        ## code goes here
```

The last four parameters are optional, but in practice they are always passed by the top level script.

- `tempdir` is the directory into which files and directories for unpacking should be created. If it is `None` a new temporary directory should be created.
- `blacklist` is a list of byte ranges that should not be scanned. If the current scan needs to blacklist a byte range it should add it to this list after finishing a scan.
- `offsets` is a dictionary containing a mapping from an identifier to a list of offsets in the file where these identifiers can be found. This list is filled by the scan `genericMarker` which always runs before anything else.
- `scanenv` is an optionally empty dictionary of environment variables that can be used to pass extra information to the pre-run method.
- `debug` is an environment variable that can be used to optionally set the scan in debugging mode so it can print more information on standard error. By default it is set to `False`.

Return values are:

- the name of a directory, containing files that were unpacked.
- the blacklist, possibly appended with new values
- a list of tags, in case any tags were added, or an empty list

Most scans have been split in two parts: one part is for searching the identifiers, correctly setting up temporary directories and collecting results. The other part is doing the actual unpacking of the data and verification.

The idea behind this split is that sometimes functionality is shared between two scans. For example, `unpackCpio` is used by both `searchUnpackCpio` and `unpackRPM`.

C.3.2 Adding an identifier for a file system or compressed file

Identifiers for new file systems and compressed files are, if available, added to `fsmagic.py` in the directory `bat`. These identifiers will be available in the `offsets` parameter that is passed to a scan, if any were found.

Good sources to find identifiers are `/usr/share/magic`, documentation for file systems or compressed files, or the output of `hexdump -C`.

C.3.3 Blacklisting and priorities

In BAT blacklists are used to prevent some scans from running on a particular byte range, because other scans have already covered these bytes, or will cover them.

The most obvious example is the `ext2` file system: in a normal setup (no encryption) it is trivial to see the content of all the individual files if an `ext2` file system image is opened. This is because this file system is mostly a concatenation of the data, with some meta data associated with the files in the file system.

If another compressed file is in the `ext2` file system it could be that it will be picked up by BAT twice: once it will be detected inside the `ext2` file system and once after the file system has been unpacked by the `ext2` file system unpacker.

Other examples are:

- cpio (files are concatenated with a header and a trailer)
- TAR (files are concatenated with some meta data)
- RPM (files are in a compressed archive with some meta data)
- ar and DEB
- some flavours of cramfs
- ubifs

To avoid duplicate scanning and false positives it is therefore necessary to prevent other scans from running on the byte range already covered by one of these files.

In BAT this is achieved by using blacklists. All unpackers have a parameter called `blacklist` which is consulted every time a file is unpacked. If a file system offset is in a blacklist the scan could use the next offset, or skip scanning the entire file, depending on the scan.

The blacklist is set for every file individually and is initially empty. If a scan is successful it adds a byte range to the blacklist. Subsequent scans will skip the byte range added by the scan.

The scans are run in a particular order to make the best use of blacklists. The order of scans is determined by the `priority` parameter in the configuration file. The file systems and concatenated files mentioned above have a higher priority and are scanned earlier than other scans that could also give a match. It is not a fool proof system, but it seems to work well enough.

C.4 Leaf scans

After everything has been unpacked each file, including the files from which other files were carved, will be scanned by the leaf scans.

C.4.1 Writing a leaf scan

The leaf scans have a simple interface. There are six parameters passed to the scan, namely the absolute path of the file, the tags of the file, an optional blacklist with byte ranges that should not be scanned, an optional list of environment

variables and an optional name of a directory for writing temporary results. For example:

```
def leafScan(path, tags, blacklist=[], scanenv={},
                debug=False, unpacktempdir=None):
        ## code goes here
```

There are no restrictions on the return values of the leaf scan, except if nothing could be found (in which case `None` is usd as return value). The result value is a tuple with a list of tags as well as one of the following:

- `None` if nothing can be found
- simple values (booleans, strings)
- custom data structure. Code that processes this data should know about its structure.

There is no restriction on the code that is run as part of the leaf scan and basically anything can be done. In BAT there are for example checks that invoke other external programs to discover dynamically linked libraries using `readelf`, find the license of a kernel module using `modinfo` or simple checks for the presence of strings in the binary that indicate the use of certain software.

The simplest scans are the ones that search for hardcoded strings. These strings are frequently found just in the package for which the check is written for. For example, the following strings can often be found in copies of the `iptables` binary and the related `libiptc` library:

```
markerStrings =
     [ 'iptables who? (do you need to insmod?)'
     , 'Will be implemented real soon.  I promise ;)'
     , 'can\'t initialize iptables table `%s\': \%s'
     ]
```

Although searching for hardcoded strings is very fast, this method has some drawbacks:

- a binary sometimes does not have these exact strings embedded
- this method will only find the strings that are hardcoded and not any other significant strings
- if another package includes the string, it will be a false positive

The quick checks should therefore only be used as an indication that further inspection of the binary is needed. A much better method is the ranking method that is also available in BAT, but which requires a special setup with a database.

C.4.2 Pretty printing for leaf scans

Pretty printing for unpackers is standardized but for leaf scans there is more flexibility. This is needed because in some cases the result as returned by the leaf scan needs post processing due to use of custom data structures.

A pretty printer can be defined in the configuration by setting `ppoutput`. The pretty printer can be in the same module as the scanning method defined in the same section, but does not need to be. If it resides in another module it can be set using `ppmodule`. The pretty printer has two parameters: a Python datastructure as returned by the scanner (this differs per scan) and a XML root element, needed to create new XML nodes. The method is expected to return a XML node in case of success, or `None` in case of failure.

If no pretty printer is defined the value as returned by the scan will be used as the content of result tag.

C.5 Aggregators

Aggregators take all information from the entire scan process and possibly modify results.

C.5.1 Writing an aggregator

Aggregators have a strict interface:

```
def aggregateexample(unpackreports, scantempdir, topleveldir, scanenv,
                     debug=False, unpacktempdir=None)
```

- `unpackreports` are the reports of the unpackers for all files
- `scantempdir` is the location of the top level data directory of the scan
- `topleveldir` is the location of the top level directory of the scan
- `scanenv` is a dictionary of environment variables
- `debug` is an environment variable that can be used to optionally set the scan in debugging mode so it can print more information on standard error. By default it is set to `False`.
- `unpacktempdir` is the location of a directory for writing temporary files. This value is optional and by default it is set to `None`.

The aggregators should read any results of the leaf scans from the pickles on disk.

If there is any result it should be returned as a dictionary with one key. It will be assigned to the results of the top level element. Examples are: the names of files which are duplicates in an archive or firmware.

C.6 Post-run methods

Post-run methods don't change the result of the whole scanning process, but only use the data from the process. For example prettyprinting a fancy report (more advanced than the standard XML report) would be a typical post-run method.

C.6.1 Writing a post-run method

Post-run methods have a strict interface:

```
def postrunHelloWorld(filename, unpackreport, scantempdir, topleveldir,
                      scanenv={}, debug=False):
        print "Hello World"
```

- `filename` is the absolute path of the scanned file, after unpacking.
- `unpackreport` is the report of unpacking the file
- `scantempdir` is the directory that contains the unpacked data
- `topleveldir` is the top level directory containing the data directory and the directory with the per file result pickles.
- `scanenv` is an optional dictionary of environment variables
- `debug` is an environment variable that can be used to optionally set the scan in debugging mode so it can print more information on standard error. By default it is set to `False`.

The post-run methods should read any results of the leaf scans from the pickles stored on disk. Since the post-run methods don't change the result in any way, but just have side effects there is no need to return anything. Any return value will be ignored.

D Building binary packages of the Binary Analysis Tool

If you want to install BAT through the package manager of your distribution you might first need to generate packages for your distribution if none exist. For BAT there is currently support to build packages for RPM-based systems and for DEB-based systems.

D.1 Building packages for RPM based systems from releases

Building RPMs from released versions of BAT is trivial: download the SRPM files for `bat`, `bat-extratools` and `bat-extratools-java` from the BAT website and rebuild them with `rpmbuild --rebuild`.

D.2 Building packages for RPM based systems from Subversion

D.2.1 Building `bat`

Building the `bat` package is fairly straightforward.

1. Make a fresh export of BAT from Subversion
2. run the command: `python setup.py bdist_rpm`

This will create an RPM file and an SRPM file. If you need to install BAT on other versions of Fedora or on other RPM based distributions you can simply rebuild the SRPM using:

```
rpmbuild --rebuild
```

D.2.2 Building `bat-extratools` and `bat-extratools-java`

Building packages for `bat-extratools` and `bat-extratools-java` is unfortunately a bit more elaborate.

1. make a fresh export of the Subversion repository
2. change the names of `bat-extratools` and the `bat-extratools-java` directories to contain the version name of the release (for example `bat-extratools-14.0`). Make a `tar.gz` archive of the directory:

   ```
   tar zcf bat-extratools-14.0.tar.gz bat-extratools-14.0
   ```

3. run `rpmbuild` to create binary packages:

   ```
   rpmbuild -ta bat-extratools-14.0.tar.gz
   ```

D.3 Building packages for DEB based systems from releases

Currently no rebuildable packages for DEB based systems are made for releases.

D.4 Building packages for DEB based systems from Subversion

D.4.1 Building `bat`

The Debian scripts were written according to the documentation for `debhelper` found at `https://wiki.ubuntu.com/PackagingGuide/Python`.

Package building and testing is done on Ubuntu 14.04 LTS. Older versions of Ubuntu are no longer supported and its use is discouraged. This is because versions of Ubuntu older than 14.04 use a broken version of the PyDot package.

To build a `.deb` package do an export of the Subversion repository first. Change to the directory `src` and type: `debuild -uc -us` to build the package. This assumes that you will have the necessary packages installed to build the package (like `devscripts` and `debhelper`).

The build process might complain about not being able to find the original sources. In our experience it is safe to ignore this. The command will build a `.deb` package which can be installed with `dpkg -i`.

D.4.2 Building `bat-extratools` and `bat-extratools-java`

To build a .deb package do an export of the Subversion repository first. Change to the correct directories (`bat-extratools` and `bat-extratools-java` and type: `debuild -uc -us` to build the packages.

There are some dependencies that need to be installed beforehand, such as `javahelper`, `ant` and `default-jdk` for bulding `bat-extratools-java` and `zlib1g-dev`, `liblzo2-dev` and `liblzma-dev` for building `bat-extratools`. These dependencies are documented in the file `debian/control` and `debuild` will warn if these packages are missing.

E Binary Analysis Tool knowledgebase

BAT comes with a mechanism to use a database backend. The default version of BAT only unpacks file systems and compressed files and runs a few simple checks on the leaf nodes of the unpacking process.

In the paper "Finding Software License Violations Through Binary Code Clone Detection" by Hemel et. al. (ACM 978-1-4503-0574-7/11/05), presented at the Mining Software Repositories 2011 conference, a method to use a database with strings extracted from source code was described. This functionality is available in the ranking module in the file `licenseversion.py`. This code is enabled by default, but if no database is present it will not do anything.

To give good results the database that is used needs to be populated with as many packages as possible, from a cross cut of all of open source software, to prevent bias towards certain packages: if you only would have BusyBox in your database, everything would look like BusyBox.

If you don't want to spend much time on downloading and processing packages, please contact Tjaldur Software Governance Solutions for purchasing a copy of a fully prepared database at `info@tjaldur.nl`.

E.1 Generating the package list

The code and license extractor wants a description file of which packages to process. This file is hardcoded to `LIST` relative to the directory that contains all source archives. The reason there is a specific file is that some packages do not follow a consistent naming scheme. By using this extra file we can cleanup names and make sure that source code archives are recognized correctly.

The file contains four values per line:

- name
- version
- archivename
- origin (defaults to "unknown" if not specified)

separated by whitespace (spaces or tabs). An example would look like this:

```
amarok 2.3.2 amarok-2.3.2.tar.bz2 kde
```

This line says that the package is `amarok`, the version number is `2.3.2`, the filename is `amarok-2.3.2.tar.bz2` and the file was downloaded from the KDE project.

There is a helper script (`generatelist.py`) to help generate the file. It can be invoked as follows:

```
python generatelist.py -f /path/to/directory/with/sources -o origin
```

The output is printed on standard output, so you want to redirect it to a file called `LIST` (as expected by the string extraction script) and optionally sorting it first:

```
python generatelist.py -f /path/to/directory/with/sources
    -o origin | sort > /path/to/directory/with/sources/LIST
```

`generatelist.py` tries to determine the name of the package by splitting the file name on the right on a `-` (dash) character. This is not always done correctly because a package uses multiple dashes, or because it does not contain a dash. In the latter case an error will be printed on standard error, informing you that a file could not be added to the list of packages and it should be added manually.

It is advised to manually inspect the file after generating it to ensure the correctness of the package names. Packages can have been renamed for a number of reasons:

- upstream projects decided to use a new name for archives (AbiWord archives for example were renamed from `abi-$VERSION.tar.gz` (used for early versions) to `abiword-$VERSION.tar.gz`).
- a distribution has renamed packages to avoid clashes during installation and allow different versions to be installed next to eachother.
- a distribution has renamed a package. For example, Debian renamed `httpd` to `apache2`.

In these cases you need to change the names of the packages, otherwise different versions of the same package will be recorded in the database as different packages, which will confuse the rating algorithm and cause it to give suboptimal results.

Other helper scripts are `dumplist.py` which recreates a package list file from a database, and `rewritelist.py` which takes two package list files and outputs a new file with package names and versions rewritten for filenames that occur in both files. These two scripts are useful if a database needs to be regenerated, possibly with new packages.

E.2 Creating the database

The program to extract strings from sourcecode is `createdb.py`. It is not part of the standard installation of BAT, but needs to be retrieved separately from version control together with `generatelist.py`. This will be changed at some point in the future.

It parses the file generated by `generatelist.py`, unpacks the files (gzip compressed TAR, bzip2 compressed TAR, LZMA compressed TAR, XZ compressed TAR and ZIP are currently supported) and scans each individual source code file (written in C, C++, assembler, QML, C#, Java, Scala, JSP, Groovy, PHP, Python, Ruby and ActionScript) for string constants, methods, functions, variables and, if enabled, licenses using Ninka and FOSSology and copyright information using FOSSology and regular expressions lifted from FOSSology.

For the Linux kernel additional information is extracted about kernel functions and variables, module information (author, license, parameters, and so on), and kernel symbol information.

`createdb.py` can be invoked as follows:

```
python createdb.py -f
    /path/to/directory/with/files -c /path/to/configurationfile
```

The configuration file is a simple configuration file in Windows INI format. An example of a configuration file is as follows:

```
[extractconfig]
configtype = global
database = /home/bat/db/master.sqlite3
scanlicense = yes
licensedb = /home/bat/db/licenses.sqlite3
nomoschunks = 10
ninkacommentsdb = /home/bat/db/ninkacomments.sqlite3
scancopyright = yes
scansecurity = yes
securitydb = /home/bat/db/security.sqlite3
cleanup = yes
wipe = no
unpackdir = /ramdisk
```

The global section is called `extractconfig`. The field `configtype` has to be set to `global`. The field `database` is used to set the path to the main database. This parameter is mandatory: if it is not set the script will exit. The parameters `scanlicense` and `scancopyright` can be used to enable or disable license and copyright scanning (default: disabled). `licensedb` is used to set the path to the copyright and licensing database. The setting `nomoschunks` can be set to tell Nomos (the license scanner in FOSSology) how many files should be scanned at once. The default value set in the database creation script is 10. Nomos can scan multiple files at once, but it has concurrency problems (see `https://github.com/fossology/fossology/issues/396` for an explanation).

The parameter `ninkacommentsdb` can be used for setting a caching database for mapping comments to licenses, as used by Ninka. This setting is mandatory if license scanning is enabled.

The setting `scansecurity` enables extraction of security information from source code. The parameter `securitydb` points to the database file that security information should be written to. At the moment only C files are searched for security bugs.

If `cleanup` is set to `yes` (default) the temporary directory with unpacked sources will be removed. If `wipe` (default: `no`) is set to `yes` all tables and indexes

will first be dropped. The parameter `unpackdir` can be used to set a location where archives are unpacked, for example a ramdisk or SSD.

In case data for string identifiers, function names and variable names has not been changed it can be copied from another database:

```
authdatabase = /home/bat/olddb/oldmaster.sqlite3
```

One use is for example when support for a new file type has been added (for example: extraction of identifiers for Ruby was added in BAT 21) and packages need to be rescanned, but it is not necessary to extract data for all files. For now this option is explicitely disabled for the Linux kernel, as some data for the Linux kernel is extracted in a different way. In the future this will likely change.

Similarly data can be copied from an authoritive licensing and copyright database:

```
authlicensedb = /home/bat/db/checked_licenses.sqlite3
```

This setting is useful if licensing and copyright data has been scanned previously and checked, or comes from a different source than Ninka and FOSSology. Currently both licensing and copyright data is copied if this option is enabled, but this will change in the future to allow for just licensing or copyright data to be copied.

Apart from the global section there are also package specific sections to add files or to ignore files. Adding extra files can be done as follows:

```
[bash]
configtype = package
extensions = .def:C
```

The section name (in the example `bash`) is the name of the package and is used by `createdb.py` to match with a package name. The field `configtype` should be set to `package`. The only field is `extensions` which defines pairs of extensions and languages for files with package specific extensions that are interesting to scan. For example `bash` has quite a few strings that end up in binaries defined in its source tree that end on `.def`. These files are only interesting in the context of `bash`. An extension/language pair has a semicolon as a separator. Multiple pairs are separated by whitespace.

Another option is to specifically ignore files, for example:

```
[freecad]
configtype = package
blacklist = Arch_rc.py
```

Multiple files can be set in the `blacklist` parameter separated by semicolons.

E.3 License extraction and copyright information extraction

The configuration for `createdb.py` has a few options. The most important ones to consider are whether or not to also extract licenses and copyrights from the

source code files. License extraction is done using the Ninka license scanner and the Nomos license scanner from FOSSology. Copyright scanning is done using the copyright scanner from FOSSology. These options are disabled by default for a few reasons:

- extracting licenses and copyrights costs significantly more time
- there are no packages for Fedora and Debian/Ubuntu for Ninka

If you want to enable license extraction, you will have to install Ninka first and change one hardcoded path that points to the main Ninka script in `createdb.py`. You will also have to install FOSSology (for which packages are available for most distributions).

E.4 Converting the SQLite database to PostgreSQL

The database creation script outputs the database in SQLite format. It is possible to use PostgreSQL as well. To convert the database from SQLite to PostgreSQL there is helper script called `bat-sqlitetopostgresql.py` that can help convert the database from SQLite to PostgreSQL.

A set of statements to create the database in PostgreSQL can be found in the files `maintenance/postgresql-table.sql` and `maintenance/postgresql-index.sql` that can be directly passed to PostgreSQL's `psql` program. Configuring PostgreSQL is out of scope of this manual.

At the moment some of the settings in the conversion script, table and index definitions are hardcoded and specific to Tjaldur Software Governance Solutions. This will be changed in the future. Please note that a few settings are hardcoded in the table and index definitions.

E.5 Database design

Depending on if the database is stored in PostgreSQL or SQLite the database tables might be in one database (PostgreSQL) or separate files (SQLite).

The main database currently has 16 tables, 9 of which are Linux kernel specific. One other table is optional:

- `processed`
- `processed_file`
- `extracted_string`
- `extracted_function`
- `extracted_name`
- `kernel_configuration`
- `kernelmodule_alias`
- `kernelmodule_author`
- `kernelmodule_description`

- `kernelmodule_firmware`
- `kernelmodule_license`
- `kernelmodule_parameter`
- `kernelmodule_parameter_description`
- `kernelmodule_version`
- `renames`

The optional table:

- `hashconversion`

The licenses database has 2 tables:

- `extracted_copyright`
- `licenses`

During creation an additional table `ninkacomments` is used, but this is only used to cache licensing information determined by the Ninka license scanner. It is not used otherwise.

The security database has 1 table:

- `security`

E.5.1 `processed` table

This table is to keep track of which versions of which packages were scanned. Its only purpose is to avoid scanning packages multiple times. It is not actively used in the ranking code.

It has the following fields:

- `package`: name of the package
- `version`: version of the package
- `filename`: name of the archive
- `origin`: site/origin where the archive was downloaded (optional)
- `checksum`: SHA256 checksum of the archive
- `downloadurl`: download URL of the site (optional)

E.5.2 `processed_file` table

This table contains information about of individual source code files that were scanned.

It has the following fields:

- `package`: name of the package the file is from (same as in `processed`)
- `version`: version of the package the file is from (same as in `processed`)
- `pathname`: relative path inside the source code archive
- `checksum`: SHA256 checksum of the file
- `filename`: filename of the file, without path component
- `thirdparty`: boolean (PostgreSQL)/`tinyint` (SQLite) indicating if the file is an obvious copy of a file from another package.

E.5.3 `extracted_string` table

This table stores the individual strings that were extracted from files and that could possibly end up in binaries.

It has the following fields:

- `stringidentifier`: string constant that was extracted
- `checksum`: SHA256 checksum of file the string constant was extracted from
- `language`: language the source code file was written in (mapped to a language family, such as C or Java)
- `linenumber`: line number where the string constant can be found in the source code file (if determined using using `xgettext`) or 0 (if determined using a regular expression).

E.5.4 `extracted_function` table

In this table information about C functions and Java methods is stored.

- `checksum`: SHA256 checksum of the file
- `functionname`: function name or method name that was extracted
- `language`: language the source code file was written in (mapped to a language family, such as C or Java)
- `linenumber`: line number where the function/method can be found in the source code file (if determined using using `xgettext`) or 0 (if determined using a regular expression).

E.5.5 `extracted_name` table

This table stores information of various names extracted from source code. Included are variable names (C), field names (Java) and class names (Java) and Linux kernel variable names.

It has the following fields:

- `checksum`: SHA256 checksum of the file
- `name`: name of variable, field or class name that was extracted
- `type`: type (field, variable, class name, etcetera)
- `language`: language the source code file was written in (mapped to a language family, such as C or Java)
- `linenumber`: line number where the function/method can be found in the source code file (if determined using using `xgettext`) or 0 (if determined using a regular expression).

E.5.6 `extracted_copyright` table

This table stores copyright information that was extracted from files by FOSSology.

It has the following fields:

- `checksum`: SHA256 checksum of the file
- `copyright`: copyright information that was extracted
- `type`: type of information that was extracted, currently `url`, `email` or `statement`
- `offset`: byte offset in the file where the copyright statement can be found

E.5.7 `hashconversion` table

The `hashconversion` table is used as a lookup table to translate between different hashes and use these for checks or reporting. The table has the following mandatory field:

- `sha256`: SHA256 checksum of the file

Any other hashes (limited to values that Python's hashlib supports, as well as CRC32 and TLSH) listed in `extrahashes` in the database creation script configuration file will be added as columns to this database. Tjaldur Software Governance Solutions by default sets MD5, SHA1, CRC32 and TLSH, which the convertor from SQLite to PostgreSQL expects to find as well, in that order.

E.5.8 `kernel_configuration` table

The Makefiles in the Linux kernel configuration contain a lot of information about which configuration includes which files. This information can be used to reconstruct a possible kernel configuration that was used to create the Linux binary image. The table has the following fields:

- `configstring`: configuration directive in Linux kernel
- `filename`: filename/directory name to which the configuration directive applies
- `version`: Linux kernel version

E.5.9 `kernelmodule_alias` table

This table is used to store information about Linux kernel module aliases. This information is declared in the Linux kernel source code using the `MODULE_ALIAS` macro. The table has the following fields:

- `checksum`: SHA256 checksum of the file
- `modulename`: name of the source code file
- `alias`: contents of the `MODULE_ALIAS` macro

E.5.10 `kernelmodule_author` table

This table is used to store information about Linux kernel module author(s). This information is declared in the Linux kernel source code using the `MODULE_AUTHOR` macro. The table has the following fields:

- `checksum`: SHA256 checksum of the file
- `modulename`: name of the source code file
- `author`: contents of the `MODULE_AUTHOR` macro

E.5.11 `kernelmodule_description` table

This table is used to store information about Linux kernel module descriptions. This information is declared in the Linux kernel source code using the `MODULE_DESCRIPTION` macro. The table has the following fields:

- `checksum`: SHA256 checksum of the file
- `modulename`: name of the source code file
- `description`:

E.5.12 `kernelmodule_firmware` table

This table is used to store information about Linux kernel module firmware. This information is declared in the Linux kernel source code using the `MODULE_FIRMWARE` macro. The table has the following fields:

- `checksum`: SHA256 checksum of the file
- `modulename`: name of the source code file
- `firmware`: contents of the `MODULE_FIRMWARE` macro

E.5.13 `kernelmodule_license` table

This table is used to store information about Linux kernel module licenses. This information is declared in the Linux kernel source code using the `MODULE_LICENSE` macro. The table has the following fields:

- `checksum`: SHA256 checksum of the file
- `modulename`: name of the source code file
- `license`: contents of the `MODULE_LICENSE` macro

E.5.14 `kernelmodule_parameter` table

This table is used to store information about Linux kernel module parameters. This information is declared in the Linux kernel source code using the `MODULE_PARM` and `module_param` macros, as well as variations of the `module_param` macro. These different notations were used for different versions of the Linux kernel and both formats have been used in the kernel at the same time. The table has the following fields:

- `checksum`: SHA256 checksum of the file
- `modulename`: name of the source code file
- `paramname`: name of the parameter
- `paramtype`: type of the parameter, as specified in the source code (various formats have been used)

E.5.15 `kernelmodule_parameter_description` table

This table is used to store information about Linux kernel module parameters descriptions. This information is declared in the Linux kernel source code using the `MODULE_PARM_DESC` macro. The table has the following fields:

- `checksum`: SHA256 checksum of the file
- `modulename`: name of the source code file
- `paramname`: name of the parameter
- `description`: descriptio of the parameter

E.5.16 `kernelmodule_version` table

This table is used to store information about Linux kernel module versions. This information is declared in the Linux kernel source code using the `MODULE_VERSION` macro. The table has the following fields:

- `checksum`: SHA256 checksum of the file
- `modulename`: name of the source code file
- `version`: contents of the `MODULE_VERSION` macro

E.5.17 `licenses` table

This table stores the licenses that were extracted from files using a source code scanner, like Ninka or FOSSology. If a file has more than one licenses there will be multiple rows for a file. It has these fields:

- `checksum`: SHA256 checksum of the file
- `license`: license as found by the scanner
- `scanner`: scanner name. Currently only Ninka and FOSSology are used in BAT, but is not limited to that: the scanner could also be a person doing a manual review.
- `version`: version of scanner. This is useful if there is for example a bug in a scanner, or to compare results from various versions.

E.5.18 `renames` table

This is a lookup table to deal with packages that have been cloned or renamed and should be treated as another package when scanning. Examples are packages in Debian that have been renamed for trademark reasons (Firefox is called Iceweasel), forks (KOffice versus Calligra), and so on.

- `originalname`: name the package was published under
- `newname`: name that the package name should be translated to

The script `clonedbinit.py` in the `maintenance` directory generates a minimal translation database. In several scans this database can be used by setting the `BAT_CLONE_DB` parameter.

E.5.19 `security_cert` table

This table stores security information that was extracted from files. It has these fields:

- `checksum`: SHA256 checksum of the file
- `securitybug`: identifier for a security bug, for example identifiers for the CERT secure coding standard.
- `linenumber`: line number where the security bug can be found

- `whitelist`: boolean value indicating whether or not the bug can safely be ignored. The idea is that this can be set by security reviewers if the security bug cannot be triggered to lower the amount of false positives.

E.5.20 `security_cve` table

This table stores information about relations between paths and CVE numbers.

- `checksum`: SHA256 checksum of the file
- `cve`: CVE identifier

E.5.21 `security_password` table

This table stores information about relations between hashes and derived passwords.

- `hash`: hash value as found in password or shadow file
- `password`: password found with a password cracker

F Identifier extraction and ranking scan

As explained identifying binaries works in two phases: first identifiers are extracted from the binaries, then the identifiers are processed by one or more scans, for example the ranking scan.

Apart from making it possible to process the identifiers with various methods there is another reason that the code is split in two parts and that is performance: extracting identifiers is very quick and can be done in parallel for many files. Computing a score can be quite expensive to do for certain files (such as a Linux kernel image). Processing identifiers *per file* in parallel instead of processing files in parallel turns out to be much faster. This is why the current ranking scan(s) are all aggregate scans and not leaf scans.

F.1 Configuring identifier extraction

```
[identifier]
type        = leaf
module      = bat.identifier
method      = searchGeneric
envvars     = BAT_NAMECACHE_C=/home/bat/db/functioncache_c:
              DEX_TMPDIR=/ramdisk:BAT_STRING_CUTOFF=5
noscan      = text:xml:graphics:pdf:compressed:
              resource:audio:video:mp4:vimswap:timezone:ico
description = Classify packages using advanced ranking mechanism
enabled     = yes
setup       = extractidentifiersetup
priority    = 1
```

The three parameters are:

- BAT_NAMECACHE_C - in case the binary is a Linux kernel image or Linux kernel module the identifier extraction scan needs to be able to look up kernel function names to filter these out and assign to the function results. For this a caching database with function names is needed. If the database backend is configured to be PostgreSQL, then the value of this parameter will be ignored, but it is (at the moment) still required to set this parameter.

- BAT_STRING_CUTOFF - this value is the mimimal length of the string that is matched (default value is 5). If extracted strings are shorter than this value they will be ignored. It is important to keep this parameter in sync with the minimum length of strings in the database extract script.

- DEX_TMPDIR - set the location of a temporary location for unpacking Android DEX files. This can be for example set to the location of a ramdisk.

F.2 Configuring the ranking method

The ranking method can be found in `bat/licenseversion.py`. The ranking method looks up strings in the database, optionally aggregates results for Java class files at the JAR level, determines versions and licenses while also removing unlikely versions from the result set.

The ranking method uses a few tables. Depending on the database backend the tables will either be in a single database (PostgreSQL) or possibly in multiple files (sqlite).

For the first part (determining which package a string belongs to) it uses tables with caching information for string constants, function names, variable names and so on. These caching tables contain a subset of information to vastly speed up scanning. There is no script in the standard distribution of BAT to create these caching tables. The second part (determining versions and licenses) other tables are used.

When the database backend is set to sqlite the configuration will be checked to see what the locations of the database files are. When the database backend is set to PostgreSQL the parameters for the database files will be ignored, but they still need to be supplied (for now).

The location of the SQLite database files can be set in the configuration file in the `envvars` option:

```
[versionlicensecopyright]
type        = aggregate
module      = bat.licenseversion
method      = determinelicense_version_copyright
noscan      = text:xml:graphics:pdf:audio:video:mp4
envvars     = BAT_DB=/home/bat/db/master.sqlite3:
              BAT_LICENSE_DB=/home/bat/db/licenses.sqlite3:
              BAT_CLONE_DB=/home/bat/db/clonedb.sqlite3:
              BAT_STRINGSCACHE_C=/home/bat/db/stringscache_c:
              BAT_STRINGSCACHE_JAVA=/home/bat/db/stringscache_java:
              BAT_NAMECACHE_C=/home/bat/db/functioncache_c:
              BAT_NAMECACHE_JAVA=/home/bat/db/functioncache_java:
              BAT_STRING_CUTOFF=5:AGGREGATE_CLEAN=1:USE_SOURCE_ORDER=1:
```

```
                BAT_RANKING_LICENSE=1:BAT_RANKING_VERSION=1:
                BAT_KEEP_VERSIONS=10:BAT_KEEP_MAXIMUM_PERCENTAGE=50:
                BAT_MINIMUM_UNIQUE=10
enabled     = yes
priority    = 3
```

The main database with all information except license information is set using BAT_DB. This option is mandatory. If it is not supplied scanning with the determining versions and licenses will be disabled.

In the database the strings, averages, function names, variable names, etcetera are split per language family (C, Java, C#, and so on). The reason for this is that strings/function names that are very significant in one programming language family could be very generic in another programming language family and vice versa. During scanning a guess will be made to see which language the program was written in and the proper caching database will be queried.

Since there are relatively few binaries (at least on Linux) that combine code from both languages the caching databases are split. This makes the caching databases a lot smaller so they can easier fit into memory. There are of course programs with language embeddeding and better support for these will be added in the future.

The names of the caching databases start with BAT_STRINGSCACHE and BAT_NAMECACHE and are postfixed with an underscore and the name of the programming language family. The strings cache database for Java for example is configured using the environment variable BAT_STRINGSCACHE_JAVA.

An optional database to deal with copied and renamed packages can be set with BAT_CLONE_DB. If set and populated the ranking scan will use information from this database to rewrite package names. This is useful if a package was renamed for a reason and different packages should be treated as if they were a single package. Examples are Ethereal that had to be renamed to Wireshark, or KOffice that was forked into Calligra, after which development on KOffice effectively stopped and everyone moved to Calligra.

The license database can be set with BAT_LICENSE_DB. If it is not supplied licensing information will not be used during the scan. If BAT_RANKING_LICENSE is not set to 1 no license information will be extracted. If BAT_RANKING_VERSION is not set to 1 no version information will be extracted. If BAT_RANKING_LICENSE is set to 1 it automatically sets BAT_RANKING_VERSION to 1 as well.

The parameter USE_SOURCE_ORDER can be used to tell the matching algorithm to assume that identifiers in the binary code are similar as in the source code and that the compiler has not reordered these. As compilers often keep the order this assigns more strings to packages. As soon as compilers start reordering identifiers this method will not work. The default setting is to not use the order of identifiers.

The parameter BAT_STRING_CUTOFF indiciates the mimimal length of the string that is matched (default value is 5). If extracted strings are shorter than this value they will be ignored. It is important to keep this parameter in sync with the minimum length of strings in the database extract script.

Results of Java class files are aggregated per JAR where the class files were found in. If the parameter AGGREGATE_CLEAN is set to 1 the class files will be removed from the result set after aggregating the results. By default class files will not be removed.

The parameters BAT_KEEP_VERSIONS, BAT_MINIMUM_UNIQUE and BAT_KEEP_MAXIMUM_PERCENTAGE are used to tell the pruning methods how many versions to keep, how many unique strings minimally should be found, and so on.

F.2.1 Interpreting the results

There are two ways to interpret the results. The recommended way is to load the result file into the graphical user interface. The other way is to have BAT pretty print the result in XML and further process the XML file.

The results of the scan can be found in the element <ranking>. This element contains:

- number of lines that were extracted from the binary
- number of lines that could be matched exactly with an entry in the database
- result per package which are a possible match

Per package the following is reported:

- name of the package
- all unique matches (strings that can only be found in this package)
- relative ranking
- percentage of the total score

For example, take the results of a run on a BusyBox binary:

```
<ranking>
  <matchedlines>1314</matchedlines>
  <extractedlines>3147</extractedlines>
  <package>
    <name>busybox</name>
    <uniquematches>
        <unique>%d heads, %d sectors/track, %d cylinders</unique>
        ...
    </uniquematches>
    <rank>1</rank>
    <percentage>98.3386895181</percentage>
  </package>
  ...
</ranking>
```

About 98% of the total score was for BusyBox, so it is a clear match. In programs were two or more packages are embedded percentages will be distributed in a different, more uniform, way.

G BusyBox script internals

The BusyBox processing scripts look simple, but behind the internals are a bit hairy. Especially extracting the correct configuration is not trivial.

G.1 Detecting BusyBox

Detecting if a binary is indeed BusyBox is trivial, since in a BusyBox binary there are almost always clear indication strings if BusyBox is used (unless they it was specifically altered to hide the use of BusyBox).

A significant set of strings to look for is:

```
BusyBox is a multi-call binary that combines many common Unix
utilities into a single executable.  Most people will create a
link to busybox for each function they wish to use and BusyBox
will act like whatever it was invoked as!
```

Another clear indicator is a BusyBox version string, for example:

```
BusyBox v1.15.2 (2009-12-03 00:14:42 CET)
```

As an exception a BusyBox binary configured to include just a single applet will not contain contain the marker strings, or the BusyBox version string. In such a case a different detection mechanism will have to be used, for example the ranking code as used by `bat-scan`, although this will only be necessary in a very small percentage of cases, since the vast majority of BusyBox instances include more than one applet.

G.2 BusyBox version strings

The BusyBox version strings have remained fairly consistent over the years:

```
BusyBox v1.00-rc2 (2006.09.14-03:08+0000) multi-call binary
BusyBox v1.1.3 (2009.09.11-12:49+0000) multi-call binary
BusyBox v1.15.2 (2009-12-03 00:14:42 CET)
```

The time stamps in the version string are irrelevant, since they are generated during build time and are not hardcoded in the source code.

Extracting version information from the BusyBox binary is not difficult. Using regular expression it is possible to look for `BusyBox v` which indicates the start of a BusyBox version string. The version number can be found immediately following this substring until ` (` (including leading space) is found.

Apart from reporting, the BusyBox version number is also used for other things, such as determining the right configuration format and accessing a knowledgebase of known applet names extracted from the standard BusyBox releases from `busybox.net`.

G.3 BusyBox configuration format

During the compilation of BusyBox a configuration file is used to determine which functionality will be included in the binary. The format of this configuration file has changed a few times over the years. Early versions used a simple header format file, with GNU C/C++ style defines. Later versions, starting 1.00pre1, moved to Kbuild, the same configuration system as used by for example the Linux kernel or OpenWrt. This format is still in use today (BusyBox 1.20.0 being the latest version at the time of writing).

Each configuration directive determines whether or not a certain piece of source code will be compiled and up in the BusyBox binary. This source code can either be a full applet, or just a piece of functionality that merely extends an existing applet.

G.4 Extracting a configuration from a BusyBox binary

Extracting the BusyBox configuration from a binary is not entirely trivial. There are a few methods which can be used:

1. run `busybox` (on a device, or inside a sandbox) and see what functionality is reported. This is probably the most accurate method, but also the hardest, since it requires access to a device, or a sandbox that has been properly set up, with all the right dependencies, and so on.

   When running `busybox` without any arguments, or with the `--help` parameter it will output a list of functions that are defined inside the binary:

   ```
   Currently defined functions:
           ar, cal, cpio, dpkg, dpkg-deb, gunzip, zcat
   ```

   These can be mapped to a configuration, using information extracted from BusyBox source code about which applets map to which configuration option.

2. extract the configuration from the binary by searching for known applet names in the firmware. The end result is the same as a previous step, but possibly with less accuracy in some cases but it is the only feasible solution if you only have a binary.

The BusyBox binary has a string embedded for every applet that is included. This is the string that is printed out if `--help` is given as a parameter to an invocation of `busybox`.

Using information about the configuration extracted from BusyBox source code these strings can be mapped to a configuration directive and a possible configuration can be reconstructed.

Depending on how the binary was compiled this can be trivial, or quite hard.

G.4.1 BusyBox linked with uClibc

In binaries that link against uClibc (a particular C library) the name of the main function of the applet is sometimes (but not always) included in the `busybox` binary as follows (a good way is to run `strings` on the binary and look at the output).

```
wget_main
```

This string maps to the name of the main function for the `wget` applet (`networking/wget.c`):

```
int wget_main(int argc, char **argv) MAIN_EXTERNALLY_VISIBLE;
```

The BusyBox authors are pretty strict in their naming and usually have a configuration directive in the a specific format (`CONFIG-$appletname`) in the Makefile, like:

```
lib-$(CONFIG_WGET)          += wget.o
```

(example taken from `networking/Kbuild` in BusyBox 1.15.2). There are cases where the format could be slightly different.

G.4.2 BusyBox linked with glibc & uClibc exceptions

Sometimes the method described in the previous section does not work for binaries that are linked with uClibc. It also does not work with binaries compiled with glibc.

If the binary is unstripped and the binary still contains symbol information it is possible to extract the right information using `readelf` (part of GNU binutils) in a similar fashion as the earlier described method.

In case there is no information available it is still possible to search inside the binary for the applet names. Because most instances of BusyBox that are installed on devices have not been modified the list of applets in the stock version of BusyBox serves as an excellent starting point.

The list as printed by `busybox` if the `--help` parameter is given is embedded in the binary. The applet names are alphabetically sorted and separated by NUL characters.

By searching for this list and splitting it accordingly it is possible to get the list of all applets that are defined. The only caveats are that a new applet that was added appears alphabetically before any of the applets that can be recognized using a list of applet names extracted from the source code, or it appears alphabetically after the last one that can be recognized.

G.5 Pretty printing a BusyBox configuration

Pretty printing a BusyBox configuration is fairly straightforward, but there are a few cases where it is hard to make a good guess:

1. aliases
2. functionality that is added to an applet, depending on a configuration directive
3. applets that use non-standard configuration names (like `CONFIG_APP_UDHCPD` instead of `CONFIG_UDHCPD` in some versions of BusyBox)
4. features

For some applets aliases are installed by default as symlinks. These aliases are recorded in the binary, but there is no separate applet for it. In the BusyBox sources (1.15.2, others might be different) these are defined as:

```
IF_CRYPTPW(APPLET_ODDNAME(mkpasswd, cryptpw, _BB_DIR_USR_BIN,
    _BB_SUID_DROP, mkpasswd))
```

So if the `cryptw` tool is built, an additional symlink called `mkpasswd` is added during installation.

If extra functionality is added to an applet in BusyBox it is defined in the source code by macros like the following:

```
IF_SHA256SUM(APPLET_ODDNAME(sha256sum, md5_sha1_sum, _BB_DIR_USR_BIN,
    _BB_SUID_DROP, sha256sum))
IF_SHA512SUM(APPLET_ODDNAME(sha512sum, md5_sha1_sum, _BB_DIR_USR_BIN,
    _BB_SUID_DROP, sha512sum))
```

The above configuration tells to add extra symlinks for `sha256sum` and `sha512sum` if BusyBox is configured for suppport for the SHA256 and SHA512 algorithms. The applet that implements this functionality is `md5_sha1_sum`.

Non-standard configuration names can be fixed by using a translation table that translates to the non-standard name. The current code has a translation table for BusyBox 1.15 and higher.

Detecting features is really hard to do in a generic way. In most cases it will even be impossible, because there are no clear markers (strings, applet names) in the binary that indicate that a certain feature is enabled. In cases there are clear marker strings these would still need to be linked to specific features. One possibility would be to parse the BusyBox sources and link strings to features, for example (from BusyBox 1.15.3, `editors/diff.c`):

```
#if ENABLE_FEATURE_DIFF_DIR
       diffdir(f1, f2);
       return exit_status;
#else
       bb_error_msg_and_die("no support for directory comparison");
#endif
```

The string `"no support for directory comparison"` only appears if the feature `ENABLE_FEATURE_DIFF_DIR` is not enabled.

Implementing this will be a lot of work and it will likely not be very useful.

G.6 Using BusyBox configurations

By referencing with information extracted from the standard BusyBox source-code it is possible to get a far more accurate configuration, because it is known which applets use which configuration, unless:

- new applets were added to BusyBox
- applets use old names, but contain different code

The names of applets that are defined in BusyBox serve as a very good starting point. How these are recorded in the sources has changed a few times and depends on the version of BusyBox. The tool `appletname-extractor.py` can extract these from the BusyBox sources and store them for later reference as a simple lookup table in Python pickle format.

Names of applets per version breakdown:

- 1.15.x and later: `include/applets.h` or `include/applets.src.h` IF syntax
- 1.1.1-1.14.x: `include/applets.h` USE syntax
- 1.00-1.1.0: `include/applets.h` (different syntax)
- 0.60.5 and earlier: `applets.h`, like 1.00-1.1.0 but with a slightly different syntax

In one particular version of BusyBox (namely 1.1.0) there is a mix of three different syntaxes: (0.60.5, 1.00 and another) for a few applets (`runlevel`, `watchdog`, `tr`).

There are also a few applets in 1.1.0 which seem to be a bit harder to detect: `busybox`, `mkfs.ext3`, `e3fsck` and `[[`. These can easily be added by hand, since there are just four of them.

Another issue that is currently unresolved is that not all the shells are correctly recognized.

G.7 Extracting configurations from BusyBox sourcecode

The `busybox.py` script makes use of a table that maps applet names to configuration directives. These tables are stored in a Python pickle and read by `busybox.py` upon startup. To generate these pickle files the `appletname-extractor.py` should be used. In the standard distribution for BAT the configurations for most versions of BusyBox are shipped.

The applet names are extracted from a file called `applets.h` or `applets.src.h`.

```
python appletname-extractor.py -a /path/to/applets.h -n $VERSION
```

The configuration will be written to a file `$VERSION-config` and should be moved into the directory containing the other configurations.

H Linux kernel identifier extraction

The `createdb.py` program processes Linux kernel source code files in a slightly different way than normal source code files. There is a lot of interesting information that can be extracted from the Linux kernel sources, as well as the binary.

There are a few challenges when working with Linux kernel source code and Linux kernel binaries. First of all there are many different variants in use and many vendors have their own slightly modified version, with extra drivers, or bug fixes from later versions, or bug fixes that might not yet have been applied to the version on `kernel.org`.

Second is that in the Linux kernel binary string constants, function names, symbols, module parameters, and so on, are intertwined and some steps need to be taken to correctly split these to avoid false positives (there are other packages where kernel function names, module parameters, symbols, and so on, are valid string constants).

H.1 Extracting visible strings from the Linux kernel binary

If a kernel is an ELF binary (sometimes) the relevant sections of the binary can be read using `readelf`. Otherwise `strings` can be run on the binary. This method will return more strings than if using `readelf`, but the extra strings are mostly extra cruft that have a low chance of matching.

H.2 Extracting visible strings from a Linux kernel module

If a kernel module is an ELF binary (most cases) the relevant sections of the binary can be read using `readelf`. Otherwise `strings` can be run on the binary. This method will return more strings than if using `readelf`, but the extra strings are mostly extra cruft that have a low chance of matching.

H.3 Extracting strings from the Linux kernel sources

The Linux kernel is full of strings that can end up in a binary. Some programmers have defined macros just specific to their part of the kernel for ease of use (often a wrapper around `printk`, other programmers use more standard mechanisms like `printk`. Most strings can be extracted from the Linux kernel using `xgettext`. A minority of strings needs to be extracted using a custom regular expression.

The following two cases are worth a closer look:

H.3.1 `EXPORT_SYMBOL` and `EXPORT_SYMBOL_GPL`

The symbols defined in the `EXPORT_SYMBOL` and `EXPORT_SYMBOL_GPL` macros end up in the kernel image. The `EXPORT_SYMBOL_GPL` symbol could be interesting for licensing reporting as well, since anything that uses this symbol should be released under the GPLv2. This is a topic for future research.

H.3.2 `module_param`

The names of parameters for kernel modules can end up in the kernel, or in the kernel module itself. The names of these parameters are typically prefixed with the name of the module (which is often, but not always) and a dot, but without the extension of the file. In cases where the module name does not match the name of the file it was defined in extra information from the build system needs to be added to determine the right string.

The code for this is in the function `__init param_sysfs_builtin` in `kernel/params.c`.

Module names are extracted from the kernel Makefiles and stored in the database together with module information (author, license, description, parameters, and so on).

H.4 Forward porting and back porting

There are some strings we scan for which might not be present in certain versions, because they were removed, or not yet included in the mainline kernel. A good example is devfs. This subsystem was removed in Linux kernel 2.6.17, but it is not safe to assume that this was done for every 2.6.17 (or later) kernel that

is out in the wild, since some vendors might have kept it and ported it to newer versions (forward porting). Similarly code from newer kernels might have been included in older versions (backporting).

H.5 Corner cases

Sometimes a `#define` or some configuration directive causes that our string matching method will not work, because the string is prepended with extra characters.

An example from `arch/arm/mach-sa1100/dma.c` from kernel 2.6.32.9:

```
#undef DEBUG
#ifdef DEBUG
#define DPRINTK( s, arg... )  printk( "dma<%p>: " s, regs , ##arg )
#else
#define DPRINTK( x... )
#endif
```

Other examples include `pr_debug`, `DBG`, `DPRINTK` and `pr_info`.
To work around this there are two ways:

1. do substring matches
2. parse the source code and record where extra code is being added as in the example above and only do substring matches in a small number of cases.

Substring matching is expensive and since it only happens in a minority of cases the second method, although not trivial to implement, would be easier. This is future work.

I Binary Analysis Tool performance tips

This section describes a few methods to increase performance of the Binary Analysis Tool, plus describe drawbacks of methods named. The standard configuration of BAT tries to be sensible, with a trade off between performance and completeness. In some cases there is quite a bit of performance to be gained by simply tweaking the configuration.

I.1 Choose the right hardware

BAT will benefit a lot from fast disk, enough memory and multiple cores. Many of the scans in BAT can be run in parallel and will scale very well (until of course disk I/O limits are reached). Invest in SSD to reduce disk I/O and more cores instead of a faster CPU. Enough memory will prevent swapping which just kills performance, especially because the ranking scan in BAT can be very I/O intensive.

I.2 Use `outputlite`

Using the default configuration the original unpacked data is not included into the result archive.

There are situations where it makes sense to include the data into the result archive, for example to make it easier to do a "post mortem" after a scan. The original data can take up a lot of space, since every original file, plus everything that might have been extracted from that file, will be included, which leads to large archives and long associated packing time.

It also has performance impact on the BAT viewer, which needs to unpack some data from the archive. The smaller the archive is, the faster unpacking is.

If the original data and the unpacked data is not relevant, then setting the option `outputlite` to `yes` in the section `[batconfig]` is highly recommended:

```
outputlite = yes
```

I.3 Do not output results in XML

By default BAT will output the results of a scan in XML. The information in here more or less matches the information that is packed in the report. If the XML file is not used for analysing resuts disabling pretty printing of the results as XML can save time, especially if there are many scanned files with many results.

Disabling the XML pretty printing can be disabled my outcommenting two directives: `module` and `output`. In the default configuration they have the following values:

```
module = bat.simpleprettyprint
output = prettyprintresxml
```

I.4 Use `AGGREGATE_CLEAN` when scanning Java JAR files

If Java JAR files are scanned then pictures and reports will be generated for each of the individual `.class` files. If only the results of the JAR file are needed, then setting `AGGREGATE_CLEAN` to `1` will prevent pictures and reports to be generated for the individual `.class` files, which can save quite some processing time and help declutter the interface as well.

Of course, not generating the pictures for individual `.class` files means that some detail might be lost, especially if there are `.class` files that contain some unexpected results.

I.5 Disable `tmp` on tmpfs

Some Linux distributions (most notably Fedora 18 and later) store the `/tmp` file system on tmpfs. This means that part of the system memory is used for the `/tmp` file system. By default on Fedora it is set to 50% of the system's memory. This could influence BAT in two ways:

1. less memory available for processing

2. BAT unpacks to `/tmp` by default, unless configured differently. If the unpack results grow big enough (which is fairly easy with big firmwares) it could fill up the partition. However, there are some external tools that will write temporary results to `/tmp`.

There are various solutions, apart from adding more memory to the machine:

- configure BAT to use another path than `/tmp` for unpacking and storing results and configure some scans in BAT to use `/tmp` or a different ramdisk (recommended)
- disable `tmp` on tmpfs (not recommended)

I.6 Use tmpfs for writing temporary results

A few scans can use tmpfs or a ramdisk to write temporary results. The scans that can benefit from this are LZMA unpacking, ranking (temporary results of DEX and ODEX unpacking), compress unpacking, JFFS2 unpacking and TAR unpacking.

J Parameter description for default scans

This section describes the default parameters for several of the scans as shipped in BAT, if not described earlier in this document. These parameters are passed to the scans as part of the environment and are defined in the `envvars` setting in the configuration file.

J.1 `compress`

The `COMPRESS_TMPDIR` parameter is used to let the scan use a different location for unpacking temporary files than the standard unpacking directory. It was introduced to let the scan unpack onto a tmpfs file system to avoid disk I/O and speed up scanning.

The `COMPRESS_MINIMUM_SIZE` parameter instructs the scan to ignore output files that are `COMPRESS_MINIMUM_SIZE` bytes in size or less. This parameter was introduced because false positives in compress unpacking are very common on Debian and Ubuntu, often leading to small sized files that contain no useful data and which could interfere with scanning.

J.2 `jffs2`

The `JFFS2_TMPDIR` parameter is used to let the scan use a different location for unpacking temporary files than the standard unpacking directory. It was introduced to let the scan unpack onto a tmpfs file system to avoid disk I/O and speed up scanning.

J.3 `lzma`

The `lzma` unpack scan has two parameters: `LZMA_MINIMUM_SIZE` and `LZMA_TMPDIR`.

The `LZMA_MINIMUM_SIZE` parameter instructs the scan to ignore output files that are `LZMA_MINIMUM_SIZE` bytes in size or less. This parameter was introduced

because false positives in LZMA unpacking are very common, often leading to small sized files that contain no useful data.

By default `LZMA_MINIMUM_SIZE` is set to 10 bytes, but this is a very conservative setting and can likely be set higher safely.

The `LZMA_TMPDIR` parameter is used to let the scan use a different location for unpacking temporary files than the standard unpacking directory. It was introduced to let the scan unpack onto a tmpfs file system to avoid disk I/O and speed up scanning.

J.4 `tar`

The `TAR_TMPDIR` parameter is used to let the scan use a different location for unpacking temporary files than the standard unpacking directory. It was introduced to let the scan unpack onto a tmpfs file system to avoid disk I/O and speed up scanning.

J.5 `xor`

The `XOR_MINIMUM` parameter is used to set the minimum amount of occurences of a key that have to be present in the file before XOR unpacking is done. This is to reduce false positives.

J.6 `file2package`

The `file2package` leaf scan has one parameter: `BAT_PACKAGE_DB`. This parameter is used to specify the location of the database used by this scan. The database can be generated using the scripts `createfiledatabasedebian.py` and `createfiledatabasefedora.py` in the subdirectory `maintenance` in the BAT source tree.

J.7 `findlibs`

For the `findlibs` aggregate scan the `ELF_SVG` parameter can be set to `1` to output the graphs in SVG format.

J.8 `findsymbols`

For the `findsymbols` aggregate scan the `KERNELSYMBOL_SVG` parameter can be set to `1` to output the graphs in SVG format. The `KERNELSYMBOL_CSV` parameter can be set to output a spreadsheet in Excel-format.

J.9 `generateimages`

The `generateimages` postrun scan has five optional parameters: `AGGREGATE_IMAGE_SYMLINK`, `BAT_IMAGEDIR`, `BAT_PICKLEDIR`, `MAXIMUM_PERCENTAGE` `MINIMUM_PERCENTAGE`

J.10 `identifier`

The `identifier` leaf scan has several parameters, including `DEX_TMPDIR`. This parameter can be used to set a location where temporary files for DEX and

ODEX (Android Dalvik files) unpacking can be written. This would typically be tmpfs or a ramdisk.

J.11 licenseversion

The licenseversion aggregate scan has a few parameters that can influence performance. One of them is AGGREGATE_CLEAN. This parameter instructs the scan to remove results for individual Java class files from the result set after aggregating results at the JAR level. Java class files that are not unpacked from a JAR file are not removed from the result set. By default this parameter is set to 0 which means that results for Java class files are not removed from the result set.

J.12 prunefiles

The prunefiles aggregate scan has two parameters: PRUNE_TAGS and PRUNE_FILEREPORT_CLEAN. The PRUNE_TAGS parameter contains a comma-separated list of tags that should be ignored and removed from the scan results. The PRUNE_FILEREPORT_CLEAN parameter can be set to indicate whether or not the result pickles for the pruned files should also be removed from disk. Example:

```
PRUNE_TAGS=png,gif:PRUNE_FILEREPORT_CLEAN=1
```

J.13 hexdump and images

The hexdump and images scans (disabled by default) have two parameters. The BAT_IMAGE_MAXFILESIZE parameter is set to specify the maximum size of a file for which a result is generated. Since output from this scan can be extremely large, and the results are not very interesting for large files it is strongly advised to cap this value.

K Default ordering of scans in BAT

BAT comes with a default configuration file. In this file an order for running the scans is specified, using the priority field: the higher the priority, the earlier the scan is run in the process. In this section the rationale behind this ordering is explained.

The order for pre-run scans, leaf scans, unpack scans and aggregate scans is described below. Since postrun scans do not change the result files and they are independent there is no order defined for them (although this might change in the future).

K.1 Pre-run scans

Most pre-run scans have the same priority, with a few exceptions, the most important being verifytext to find out if a file is ASCII only, or if there are any non-ASCII characters in the file. Since many of the scans (including pre-run scans) only work on non-ASCII files it is important to find out soon if a file contains only ASCII characters or not.

The order for pre-run scans is:

1. checkXML
2. verifytext
3. verifyjava
4. verifyelf, verifygraphics, verifysqlite3
5. verifyandroiddex, verifyandroidodex, verifyandroidresource, verifyandroidxml, verifyico, verifyjar, verifymessagecatalog, verifyogg, verifyotf, verifyttf, verifytz, verifywoff, vimswap

K.2 Unpack scans

As a general rule of thumb: compressed formats are scanned last, while simple containers that concatenate contents, or where the original content can still be (partially) recognised, are scanned first.

An example of a container is TAR: content is simply concatenated without compression. If the TAR archive would contain a file of a certain type (such as a gzip compressed file) and the unpacker for that type is run first it will try to carve it from the TAR file, blacklist the byte range, and the TAR unpacker would not successfully run.

For the compressed files on the other hand the original content isn't visible without unpacking so no other scans will pick it up and they can have a low priority.

The order that is defined starts with byteSwap, a special unpacker that is needed to unpack firmwares of certain devices, where a different kind of flash chip is used, needing bytes in a firmware to be swapped first before any other scan can be run.

Then the unpack scans for various container formats and file systems are run. The order in which they appear is not fool proof: container files could be embedded in container files with a lower priority, but BAT comes with (hopefully) sane defaults to prevent this.

Second to last unpack scans for compressed files where all data is packed in such a way that the original content can't be seen without unpacking are run,

Finally there are some scans that unpack text files (base64) or media files. The lzma unpack scan also has the lowest priority because of possibly many false positives.

The order of the unpack scans as defined in BAT 23 is:

1. byteSwap
2. tar
3. pdf_unpack, iso9660
4. cramfs, ext2fs, ubi
5. ar, cpio, java_serialized, romfs, rpm, upx, yaffs
6. exe, jffs2, squashfs
7. 7z, arj, bzip2, cab, compress, gzip, installshield, lrzip, lzip, lzo, pack200, rar, rzip, xz, zip, chm
8. base64, gif, ico, png, swf, lzma

K.3 Leaf scans

There is currently only one explicit ordering: `kernelchecks` is run before `identifier` because `identifier` depends on the result of `kernelchecks`. For the rest the order of the leaf scans does not matter.

K.4 Aggregate scans

Aggregate scans have a clear order. Reports and (most) images are generated at the very end when all information is known. Other scans are mostly independent of eachother, but are usually run before `versionlicensecopyright` to prevent having to read big report pickles from disk.

The order for pre-run scans is:

1. `fixduplicates`
2. `prunefiles` (disabled by default)
3. `findduplicates`
4. `findlibs` `findsymbols`
5. `jars`, `kernelversions`
6. `versionlicensecopyright`
7. `shellinvocations`
8. `generateimages`, `generatereports`, `generatejson`